Formal Specs Verifier ATG: a Tool for Model-based Generation of High Coverage Test Suites

Orlando Ferrante, Marco Marazza, Alberto Ferrari

Formal Specs Verifier ATG: a Tool for Model-based Generation of High Coverage Test Suites

Orlando Ferrante, Marco Marazza, Alberto Ferrari
ALES - UTSCE,
Piazza della Repubblica, 68 - 00185, Roma - Italy
e-mail: name.surname@utsce.utc.com

***Abstract*—In this paper we describe Formal Specs Verifier Automatic Test Generation, a tool generating high coverage test suites for embedded systems. Our tool implements a test case synthesis algorithm using a combination of model checking and optimization techniques starting from a Simulink/Stateflow model of the System Under Test. The main contributions of this paper are the following: we (1) give an extended description of our test generation algorithm, (2) describe the algorithm implementation as part of the Formal Specs Verifier framework, (3) present a concrete application of the tool to a cruise control case study and discuss experimental results comparing our algorithm with a state-of-the art COTS tool.**

***Keywords*—Model-based Automatic Test Generation, High Coverage Test Suites, Formal Methods**

I. Introduction

Testing complex hardware-software embedded systems' architectures is one of the most important and costly phases of their entire development life cycle. A significant portion of development time is spent indeed for the verification and validation phases (V&V) during which teams of test engineers aim at discovering requirements' misinterpretation and implementation errors. Early error detection facilitates correction complexity, which in turn favors minimizing the overall system development cost and time. Testing consists in the execution of a set of predefined input vectors on the System Under Test (SUT) and observation of its response for detecting possible deviations from the expected behavior. In order to perform the testing phase, test vectors (or test cases) must be provided to the test execution environment. A test vector consists of a sequence of pairs (*inputs*, *outputs*) where *inputs* is the set of values to be applied to the system and *outputs* is the set of expected output values. A set of test cases is said to be a *test suite*. In case the execution of the system produces a set of output values not matching the expected ones the test can be used to highlight an error of the SUT. To quantitatively evaluate the quality of test suites, hence of the entire test execution, coverage metrics are used. Coverage metrics measure the effectiveness of a test suite i.e. how much it covers: (1) the structure of the SUT, as in MC/DC coverage metrics, (2) the set of requirements, as in requirements-based testing, or (3) a meaningful subset of admissible faults of the system, as in fault-injection testing. A good test suite satisfies as much as possible a given coverage metric. As a general rule the more the tests executed, the higher the confidence about the correctness of the system, as long as the executed test cases are of high quality. However, the generation of high quality test suites has a relevant impact on costs, e.g. complexity, setup time and execution time; hence, besides providing test suites maximizing a given coverage metric (optimality), it is of paramount importance to ensure a reduced number of test cases (efficiency). Typically, a test suite represents a trade-off between optimality and efficiency. Several techniques have been adopted for the test generation process, especially in the area of test generation from system models whose main idea is to represent the SUT using a formal model and use automatic algorithms for the definition of test cases (model-based test generation). In particular, the use of model checking techniques for test generation has been extensively explored: the test generation procedure is formulated as a problem of reachability, enriching the model with test objectives. Such test objectives must be achieved by the produced test suite in order for a given coverage metric to be satisfied. Once the model is enriched with test objectives, a model checker is used to analyze their reachability. In case a test objective is reachable, a counter-example is produced and stored as a test case, which is said to *cover* that test objective. Once all test objectives are covered, the generated set of tests is a high coverage test suite. In this paper we describe a novel approach for the automatic test generation from system models combining bounded model checking and optimization to generate efficient test cases and test suites. This paper is an extension of the algorithm presented in [1]; the main contributions of current paper can be summarized as follows: we (1) provide an extended description of the test generation algorithm, (2) describe the implementation of the algorithm in our Formal Specs Verifier framework and (3) show the application of our tool to a cruise control case study, along with additional experimental results. The proposed use case only covers the MC/DC as an example. The paper is organized as follows: Section II stresses the central role played by model-based test generation and model-based testing in current industry development processes. Section III summarizes existing approaches for test case generation using exhaustive search techniques. Section IV provides a formal description of the problem our algorithm tries to address and Section VI describes our algorithm in details. Section VII provides an overview of the algorithm implementation as well as an application to a cruise control example and finally Section VIII concludes the paper.

II. Model Based Test Generation

Current practice in industrial systems' design and development process is to create *models* –also known as virtual prototypes– of the system being developed. A model is the artifact meant to imitate the system of interest. The system can be abstracted (modeled) at different levels of refinement

and each level of refinement has its own best-fitting formal models. For example, a mathematical model is a set of formal definitions and mathematical formulas that describe the system under analysis. At the various system development phases, models are gradually refined until these get detailed enough to allow for physical implementation of the system. The system implementation can be achieved by means of either fully-automated or pseudo-automated synthesis tools. With regards to testing, the main benefits of model-based approach are manifold: (1) tests can be automatically generated from models (model-based automatic test generation), (2) testing can be performed at different refinement levels, far before the system gets implemented and, more interestingly, (3) tests generated at early phases of the system design can be refined and re-used in later phases of the design to check whether all system properties are still fulfilled (back-to-back testing). Virtual prototyping and related testing are particularly important when the realization of components involves different technology suppliers and manufacturers, and requirements fulfillment must be checked by all the parties at all stages of the design process. This work targets model-based generation of test suites used to bring evidence that functional and non-functional metrics have been achieved by the design implementation.

III. Related work

Several techniques have been described in literature on the topic of automatic test generation. In this section we provide a brief summary of the available techniques pointing out the differences with the one proposed in this paper. Evolutionary and genetic approaches [2] generate tests by randomly exercising the inputs of the SUT and measuring the quality of the test maximizing an objective function. Such function is derived from a structural analysis of the SUT, hence a test that provides better values of the cost function is selected in a set of possible generation process outcomes. In addition, evolutionary testing employs evolutionary algorithm techniques for selection and generation of new tests from a previously generated set of tests. These techniques rely on random search adding structure to the search to avoid generating low coverage tests. However, there are no guarantees that a selected behavior is effectively the best test case with respect to a given coverage metric. Model-checking based techniques are heavily studied for the generation of test cases. In [3] the authors describe the use of a model checking engine for the generation of test cases starting from a modified version of the SUT model enriched with a reset transition (i.e. a variable that, when true, resets the entire state of SUT to the initial state) and test objectives related to the coverage criteria. The proposed approach differs from ours in several ways. At first, the number of satisfied objectives is a non-deterministic result of the execution of the algorithm hence each test case may cover an arbitrary number of test cases, whereas our method allows for selecting the maximum number of satisfiable test objectives at each execution. In addition, the generated tests are produced monolithically starting from the initial model not allowing the application of an incremental test generation methodology; hence, the applicability of the technique to concrete industrial size cases heavily depends on the size of the input model [4]. In [5] a methodology to generate tests from a formalization of requirements based on a tabular representation is described and the counter-example finding capabilities of model checkers is described as a technique to derive such counter-examples. The use of the model checker is straightforward and there is not any guarantee on the quality of generated tests. In [6] the authors describe a method for extracting test cases from Statecharts state machines considering state transitions covering (i.e. covering all states and/or all transitions of the Statecharts). The generation is executed by enriching the model with additional states and searching from counter-examples of a specific CTL formula. The method may produce redundant test cases and after test generation an additional test reduction phase is needed. This approach differs from ours in that it does not guarantee the generation of a minimal set of high coverage test cases and it relies on a specific CTL-based formulation of the problem. As a consequence it cannot be solved using SAT-based model checking techniques, that usually perform better than BDD-based techniques when a counter-example search is performed. In [7] a method exploiting random-simulation and formal verification is described based on a semi-formal method for the traversing of Extended Finite State Machines (EFSMs) that allows for reaching deep test cases exploiting the strength of simulation-based approaches. However, no guarantees are given in terms of efficiency of the generated suite and the use of the model checking engine is limited to the analysis of constraints to guide the main simulation-based test generation algorithm. In [8] the authors propose a test generation method based on the reachability of given test cases as well as extension of already generated test cases. This approach differs from ours in several ways. First, it does not provide guarantees that every generated test is optimal with respect to a specific coverage metric. The model checker is called in order to satisfy some of the test goals but no guarantees are provided to the number of the goals satisfied. Second, it does not employ an incremental test generation procedure: at every execution no guarantees are given about the length of the found counter-example: there might be cases in which a high coverage test case has a given length, but the same results would be obtained with a shorter test case.

IV. Problem Formulation

A. Model And Coverage Criteria Formulation

We focus on the problem of generating high coverage test suites for discrete time models formalized as connections of blocks. Several languages are available to capture embedded control systems using this formalism, e.g. MATLAB Simulink/Stateflow and Esterel SCADE[1]. A model $\mathcal{M}$ can be formally represented as a connection of blocks exposing a well-defined interface in terms of input and output ports, each describing its run-time behavior as an Extended Finite State Machine (EFSM). Block's interfaces are formalized as a pair of input and output vectors $u[k] \in X, y[k] \in Y$, where X and Y represent the vector domains and $k \in \mathbb{N}$ the discrete time. Each block behavior is then formalized as a transition function $F[u, x, f, k]$ that at each discrete time k maps the vectors input $u[k]$ and current state $x[k]$ to an output vector $y[k]$ and next-state values $x'[k]$, where $x'[k] = x[k+1]$. In order to be fully specified, an initial value for the state vector should be provided. Connections between blocks act as constraints between the values of inputs and outputs that must match at every time step. The transition relation of a block

[1] http://www.esterel-technologies.com/

contains logical and arithmetic expressions for Boolean and bit vectors. For a complete description of the richness of the Simulink/Stateflow languages please refer to [9]. A test suite Π is a set of test traces $\pi_1, \ldots, \pi_n$, each representing a finite sequence of admissible input values: $\pi_k = \langle u[1], \ldots, u[m] \rangle$. Each test trace π_k has a finite length $\|\pi_k\| = m$ that corresponds to the maximum time step for which the model is exercised. The objective of test generation is to produce a test suite that exercises the model for maximizing a given coverage criterion. Several coverage criteria have been defined depending on the test generation algorithm input. As an example for finite state machine a relevant coverage criteria is related to the capability of exercising the state machine enabling as much transitions as possible (transition coverage criteria). In model-based approaches several criteria can be defined depending on the input model. In our flow, Simulink and Stateflow models are processed and we use the following coverage criteria (that can be applied to similar languages such as Esterel SCADE). **MC/DC coverage**: following the definition of MC/DC for software a similar criterion has been followed for model elements associated to Boolean formulas. More precisely, for each block corresponding to a Boolean formula $y = f[u_1, \ldots, u_N]$ to fully satisfy the criterion there should exist a trace such that each input affects the truth value of the output independently of the other inputs [10]. **Relations coverage**: for each block that compares two values $y = u_1 \diamond u_2$ (where $\diamond \in \{\leq, <, =, \neq, >, \geq\}$), there should exist tests for which the output value y transitions from FALSE to TRUE and vice-versa. **State coverage**: for state machines, there should be tests such that the state machine states assume all possible values. **Transition coverage**: for each state machine, there should be tests such that all the transitions of the state machines are asserted.
In our flow, Simulink and Stateflow models are processed and we use the following coverage criteria (that can be applied to similar languages such as Esterel SCADE): MC/DC coverage, relations coverage, state coverage and transition coverage. The first two criteria apply also to guards and actions of the state machines; hence, if a guard is a composite Boolean expression, the test suite should provide tests covering the guard with a maximum MC/DC percentage value.

B. Problem Formulation

The test generation problem we address in this paper can be now formalized as follows. Given a discrete time model $\mathcal{M}$, a coverage metric and a test generation maximum time step m, generate a test suite $\Pi = \{\pi_1, \ldots, \pi_n\}$ of vectors of length $\|\pi_k\| \leq m$, such that:

1) the achieved coverage is *maximal* with respect to the coverage metric
2) each test trace contributes to *increase* the coverage of the test suite

Objective 1) ensures that the generated test suite achieves high test effectiveness, provides high coverage of the input model and captures as much errors as possible, whereas objective 2) ensures that the cost of test execution is well-balanced, meaning that each test trace effectively improves the overall coverage of the model and avoids execution of useless tests.

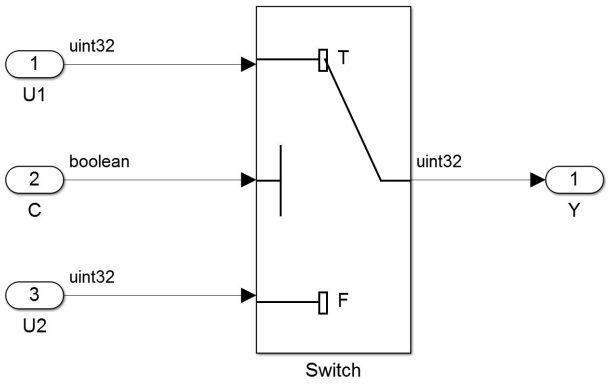

Fig. 1. Test Objective example

V. Maximal Coverage Test Case Generation

VI. Algorithm Description

In this section the test generation algorithm presented in [1] is summarized and extended showing the usage of monitor variables for the automatic synthesis of test cases. In Section VI-A the algorithm is summarized, whereas in Section VI-B the details of the efficient search sub-activity are provided. This section describes the details of the efficient search sub-activity of the test generation algorithm we presented in [1] and differs from a previous implementation [11] generating Minimal Critical Failure Sets.

A. Test Generation Flow

The overall flow of the test generation algorithm is described in Fig. 2. The initial formal model is elaborated in a model transformation step that enriches the input adding a finite number of *test objectives*. Each test objective represents a Boolean condition over the discrete time that should be satisfied by at least one generated test case. The formal description of the test objective depends on the coverage criteria selected by the user (e.g. MC/DC, relation coverage, transition coverage, etc.). During the test objectives generation step the input model is instrumented with additional Boolean expressions representing the test objectives the Automatic Test Generation (ATG) step should satisfy. Each test objective is derived by analyzing the input model blocks according to the selected criteria. As an example consider the Simulink model in Fig. 1 containing a Comparator block and a Switch block. The Comparator has an output that is true when input u_1 is greater or equal to zero and false otherwise, while the Switch connects to the output the first input when the control input c is FALSE and the second input (u_2) otherwise.
In case the user selected the *relation coverage* criteria, two Test Objectives (TOs) to be satisfied would have been derived for the Comparator block: the first TO is true when the comparator output transitions from FALSE to TRUE and the second is true when it transitions from TRUE to FALSE. Similarly, when the MC/DC criterion is selected, two additional test objectives are derived: one that is true when the control input of the Switch transitions from FALSE to TRUE and another one for the opposite condition. Using this approach we are able to convert different coverage criteria to a common criterion (test objective criterion) and the sub-sequent test generation step will try to produce a high coverage test suite maximizing the number

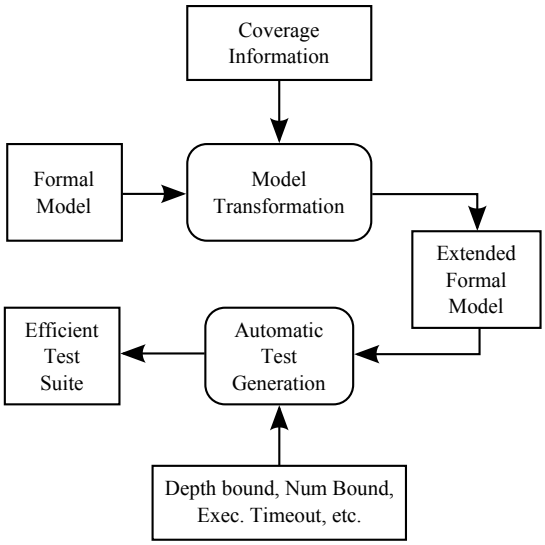

Fig. 2. Test Generation Flow

of test objectives satisfied by the generated traces. Once the extended model is produced, the Automatic Test Generation algorithm generates a set of test traces. Each trace has a finite length and satisfies the maximum number of test objectives that have not been yet satisfied by other (previously generated) test cases. As stated previously, the generated test cases verify the following properties:

- The test suite coverage percentage always increments with the addition of a new test case;
- Each test case is not redundant: no other test case of the same length covering the same (or a super-set of the) set of satisfied test objectives exists.

The ATG process can be bounded in time or in number of generated test cases by setting a set of parameters controlling its execution.

The algorithm is described in Algorithm 1. The algorithm is described in [1] and is summarized here for clarity. At start-up the explored depth bound is set to the initial value (it might be 1 or user-defined). The algorithm loops until all test objectives are satisfied or a given resource/time/depth bound is reached and in each iteration a subset of not yet satisfied test objectives (TO) is identified. The subset may be the entire set of unsatisfied TOs or it may be driven by the user needs (i.e. all the TOs related to the coverage of a given sub-function of the SUT, etc.). Once the TOs are selected an inner loop is performed until an explicit exit condition occurs or all the test objectives selected have been covered. In the inner loop, the formal engine is queried to find all the test cases of length equal to l that maximize the number of satisfied TOs among the ones selected at previous step. Each test case satisfies a unique subset of TOs (i.e. there are not two test cases satisfying the same subset of TOs). If new test cases are found, collect all the subsets of satisfied TOs and remove them from the search to avoid search again their satisfaction in next iteration. In case no new test cases are found we can assess that for the given length bound, test cases satisfying the selected test objectives do not exist. This claim is possible because of the exhaustive search

Algorithm 1 High Level View of the Test Generator Algorithm

Input: $\Theta = \{TO_1, \ldots, TO_J\}$
Input: L max explored step, T_{OUT} Algorithm execution timeout

Init
$l \leftarrow 1$, Currently Explored Length index
$\mathcal{TC}[0] \leftarrow \emptyset$, Test Case Collection @ step 0
$\Omega[0] \leftarrow \Theta$ = set of not yet satisfied TOs @ step 0
$\Psi[0] \leftarrow \emptyset$ = set of satisfied test objectives @ step 0

1: **while** $((\Psi \neq \Theta) \wedge (l \leq L) \wedge (\neg T_{OUT}))$ **do**
2: $\Omega[l]$ = SelectTestObjectives $(\Omega[l-1], \Theta)$
3: $\Psi[l] = \emptyset$
4: $bDone = false$
5: **while** $(\Omega[l] \neq \emptyset) \wedge (bDone = false)$ **do**
6: (Ψ, tc) = FindTestCaseMaxSat($\Omega[l], l$)
7: **if** $(\{tc\} \neq \emptyset)$ **then**
8: $\mathcal{TC}[l] \leftarrow \mathcal{TC}[l-1] \cup \{tc\}$
9: $\Psi[l] \leftarrow \Psi[l-1] \cup \Psi$
10: $\Omega[l] \leftarrow \Omega[l] - \Psi$
11: $bDone = false$
12: **else**
13: $l = l + 1$
14: $bDone = true$
15: **end if**
16: **end while**
17: **end while**
18: **return** $\Psi[l], \mathcal{TC}[l]$

of the underlying formal back-end. Hence the TO selector can be executed again to select a new subset of TOs or exiting from the loop if no more TOs can be selected. If during last iteration no new test cases are found, it is not possible to satisfy any of the current set of selected TOs for the given length, hence the explored depth bound is increased and a new iteration is started if resource limits are not reached (i.e. timeout, memory consumption, etc.).

1) The algorithm loops until all test objectives are satisfied or a given resource/time/depth bound is reached;
2) While this bound has not been reached:
   a) A subset of not yet satisfied test objectives (TO) is identified. The subset may be the entire set of unsatisfied TOs or it may be driven by the user needs (i.e. all the TOs related to the coverage of a given sub-function of the SUT, etc.)
   b) The following loop will be executed until new test cases are not found or the selection mechanism has not anymore TOs to select:
      i) The formal engine is queried to find all the test cases of length equal to the current length bound that maximize the number of satisfied TOs among the ones selected at previous step. Each test case satisfies a unique subset of unsatisfied TOs i.e. there are not two test cases satisfying the same subset of TOs.
      ii) If new test cases are found, collect all

the subsets of satisfied TOs and remove them from the search to avoid search again their satisfaction in next iteration.

iii) In case no new test cases are found, we can assess that for the given length bound do not exist test cases satisfying the selected test objectives. This claim is possible because of the exhaustive search of the underlying formal back-end. Hence the TO selector can be executed again to select a new subset of TOs or exiting from the loop if no more TOs can be selected.

c. If during last iteration no new test cases are found, it is not possible to satisfy any of the current set of selected TOs for the given length. Hence the explored depth bound is increased

The algorithm is guaranteed to terminate provided that the test objective selection performed at line 2 is able to identify the test objectives previously selected even if not satisfied. The simplest admissible selection mechanism picks the entire set of unsatisfied objectives at once and exits from the outer loop at point 3 after the first execution. More efficient test selection mechanisms are admissible and possible but for brevity we will not cover this topic in the report. During the search at line 6 a sub-procedure is called in order to produce queries to the formal engine and storing the counter-examples provided by the model checker as test cases. A Detailed description of the procedure is provided in Algorithm 2. The overall algorithm proceeds in an incremental fashion. Starting from an initial exploration depth bound all the test cases of a given length that maximally satisfies the test objectives are found. After the formal engine proofs that no more test cases of the given length can satisfy additional TOs, the explored bound is incremented and the search is executed again. Due the exhaustive nature of the search, it is guarantee that the set (or a super-set) of the TOs satisfied by a test case of length k cannot be satisfied by test case of length $j < k$. Hence, the method is efficient in the following sense:

- each new test case covers only new test objectives hence it strictly increments the coverage of the test suite by covering only not yet covered test objectives;
- every test case covers the maximal number of unsatisfied test objectives of a given length

In addition the incremental nature of the algorithm allows for generating test cases by starting from a complex problem in the number of TOs but simpler in the unrolling of the SUT model transition relation and incrementally increasing the complexity due the unrolling of the transition relation but reducing the number of satisfiable test objectives. Our experience with industry sized models shows that this trade off allows for applying ATG on complex models since the satisfied TOs are removed incrementally at every step and the complexity of the formal problem introduced by the increment of the explored bound is partially mitigated by the reduced number of TOs to be satisfied.

B. Maximal Coverage Test Case Generation

The mechanism to find the test case satisfying the maximum number of TOs is an important part of the optimal test suite generation procedure and in this sub-section it will be described in details. The algorithm relies on the concept of monitor variable associated to a test objective TO. A monitor is an integer variable and can be seen as a function m associated to a test objective TO that at every execution step k evolves as follows:

$$m_{TO}[k] = \begin{cases} 1 \quad if \begin{cases} m_{TO}[k-1] = 1, or \\ \exists j \leq k \text{ s.t. } TO \text{ satisfied at step } j \end{cases} \\ 0 \quad \text{otherwise} \end{cases}$$

and

$$m_{TO}[0] = \begin{cases} 1 & \text{if } TO \text{ satisfied at initial step} \\ 0 & \text{otherwise} \end{cases}$$

Each test objective T_k has an associated monitor variable m_k. Given a set of unsatisfied test objectives $T_1, T_2, \ldots, T_N$ and their associated monitors $m_1, m_2, \ldots, m_N$ the algorithm that searches for the test case set is described by the pseudo-code shown by Algorithm 2. The algorithm loops until there are counter examples found (lines $1 \ldots 9$). The search at step 2 searches for a counter example that maximizes the sum of the values of the monitors $m_1, \ldots, m_N$. The counter-example is obtained applying bounded model checking on the input model. The model checker takes into account the dynamics of the SUT and the given coverage metrics (that is used to generate the test objectives and the associated monitors). The maximization procedure ensures that the found counter-example exercises as much test objectives as possible. If a counter example is found the monitor values' configuration is extracted and a new constraint is added to the model in order to exclude the configuration from future searches (line 4, 5). This step ensures the progress of the iteration loop avoiding the model checker to find a counter-example satisfying the same test objectives over and over (there might be an infinity of them). Then the test case is extracted from the counter

Algorithm 2 Maximal Coverage Test Generator Algorithm

Input: model: The enhanced formal model under analysis
Input: $m_1, m_2, \ldots, m_N$: Set of model variables representing unsatisfied test objectives' monitors
Output: testCases : Set of produced test cases

1: **repeat**
2: find a counter example such that $m_1 + m_2 + \ldots + m_N$ is maximal
3: **if** counter example exists **then**
4: define $\mathbf{m}^* = [m_1^*, m_2^*, \ldots, m_N^*]$ the found monitor configuration
5: exclude $\mathbf{m}^*$ from the admissible solutions of the maximization search
6: extract test case values for the found counter-example
7: add extracted test case to the testCases set
8: **end if**
9: **until** counter-example has been found

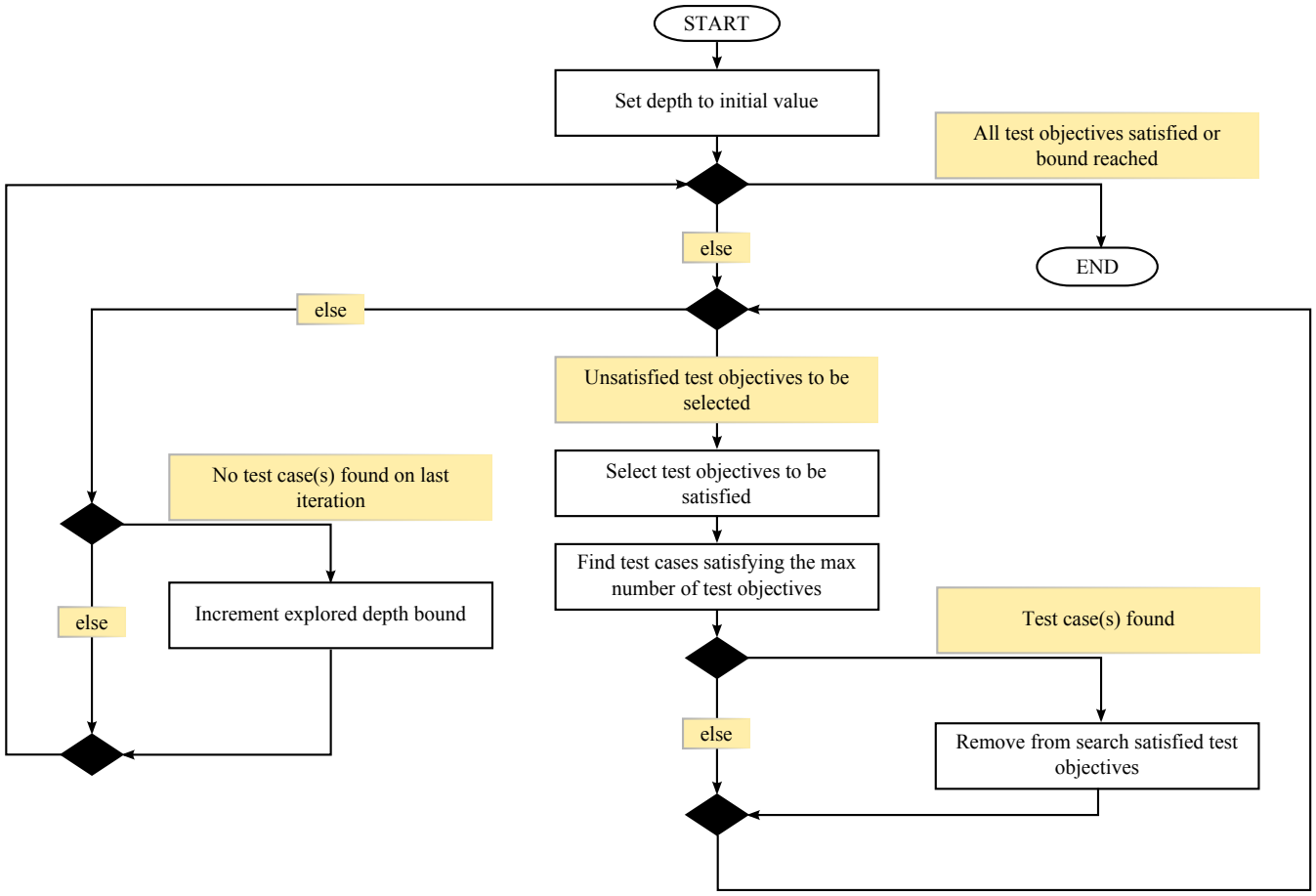

Fig. 3. Test Generation Work Flow

example by storing the values of the interesting variables of the system (inputs, outputs and internal variables) and finally it is added to the output set (line 6 and 7). The loop is then executed again looking for additional maximal possible monitor configurations until no more configurations can be found. This guarantees that: 1) every test case produced by the algorithm maximizes the number of test objectives covered and it is not possible that another test case satisfies a super set of the covered ones and 2) for the produced test suite is true by construction that there are not two test cases satisfying the same set of test objectives.

VII. Algorithm Implementation in FSV

Our automatic test generation algorithm has been implemented in the FormalSpecs Verifier (FSV) framework for the verification of embedded systems.

A. *FormalSpecs Verifier*

The FormalSpecs Verifier is a framework targeting complex embedded systems verification. The core of the tool is based on a translator from Simulink modeling language to NuSMV native language. The transformation process produces a semantically equivalent NuSMV representation of the input model taking into account the non-determinism resolution that may be introduced during the transformation step. In Fig. 4 the generic flow is described in details. As a first step the Simulink textual

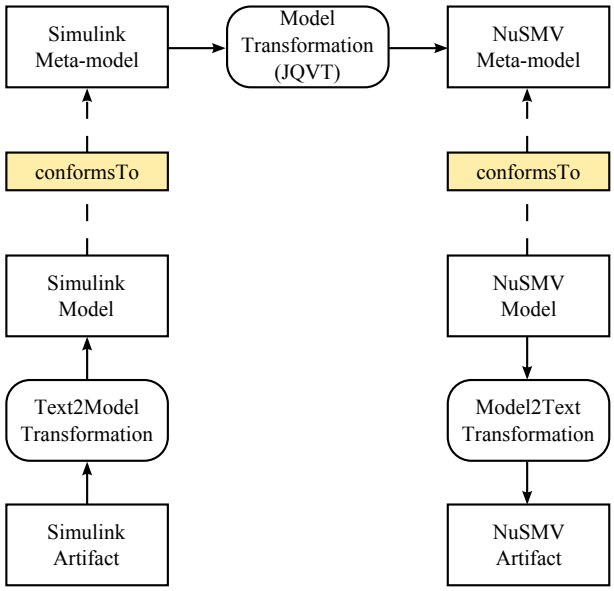

Fig. 4. Formal Specs Verifier Model Transformation Flow

file is parsed. Then the parsed Simulink model is processed generating a semantically equivalent NuSMV model that is used to generate the concrete NuSMV artifact with a model to text step. The technology used to perform the model trans-

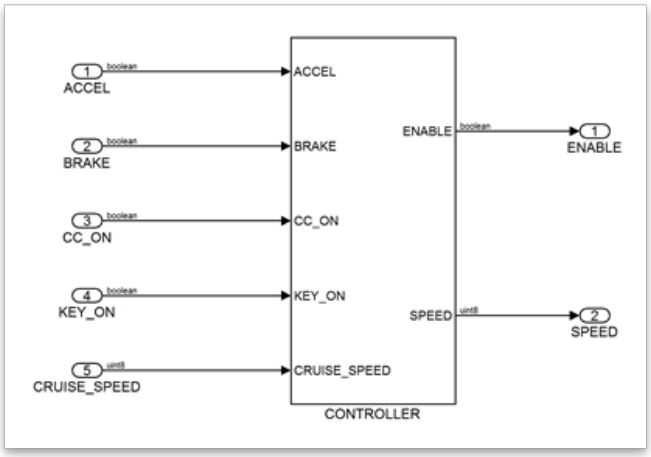

Fig. 5. Controller sub-system

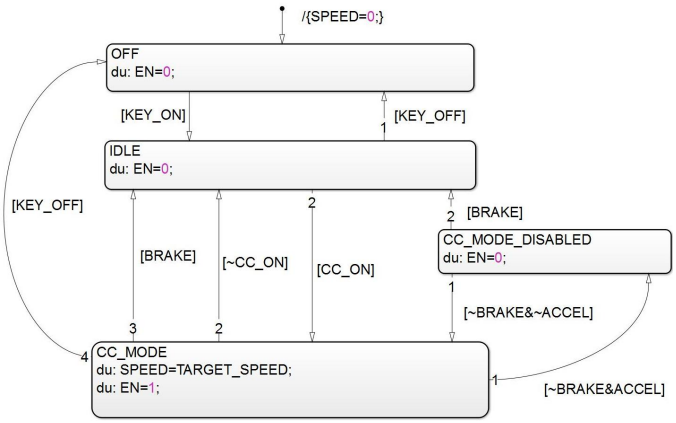

Fig. 6. Controller Extended Finite State Machine

formation step is an internally developed Java embodiment of the OMG Query/View/Transformation (QVT) language called JQVT. The JQVT library aims at providing an industry-level operational implementation of the QVT language. It supports the definition of QVT mappings and the definition of mappings inheritance, disjunction and merging. JQVT allows capturing the mapping relation that links a source model element to a target model element and it supports the *resolve* and *resolveIn* operators to retrieve the set of mapping source model elements from a given mapped target model element. JQVT does not support the entire QVT specification. However, it has been extensively used as translation infrastructure of different tools for the translation of industry-level sized models [12].

B. Cruise Control Example

We show the application of our test generation algorithm to a cruise control reference example implemented in MATLAB Simulink. We consider a modified version of the model proposed by Aldrich in [13]. A cruise control is an embedded system that regulates the speed of a vehicle based on a set of commands provided by the driver; the interface of the control algorithm appears to the system as represented in Fig. 5. The ACCEL and BRAKE inputs are Boolean values representing the pressure of the accelerator and brake pedal by the vehicle driver. The CC_ON Boolean input is set when the driver wants to engage the cruise control. The KEY_ON input represents the presence of the key (Boolean value) and finally the CRUISE_SPEED value is an unsigned integer input set by the user representing the desired cruise speed. The outputs of the controller are the ENABLE Boolean value that is true whenever the cruise controller is actively controlling the vehicles speed and the SPEED unsigned integer value representing the reference speed passed to the cascade speed controller that acts directly on the engine throttle based on the reference and current speed values. In Fig. 6 the internal modal logic of the controller is represented as an extended finite state machine (Simulink Stateflow machine).
The controller is initially in an OFF state and passing thru an IDLE state can go on a controller active mode (CC_MODE) or disengaged mode (CC_MODE_DISABLED) in which it returns the direct control to the driver. This happens whenever

```
INFO : ------------------------------------------------------------
INFO :  Transformation Statistics
INFO : ------------------------------------------------------------
INFO :           79 Condition   Test Objectives
INFO :          112 MC/DC       Test Objectives
INFO :
INFO :          191 Total       Test Objectives
INFO : ------------------------------------------------------------
INFO :  Transformation Summary
INFO : ------------------------------------------------------------
INFO :           30 Translated     Modules
INFO :            0 Ignored        Modules
INFO :            2 Not translated Modules
INFO :
INFO :           32 Total          Modules
INFO : ------------------------------------------------------------
INFO :  Error Summary
INFO : ------------------------------------------------------------
INFO :       0   warnings
INFO :       0   errors
INFO : ------------------------------------------------------------
```

Fig. 7. Automatic Generation of Test Objectives Results

she/he interacts with the car pressing the acceleration or brake pedal. The FormalSpecs Verifier Automatic Test Generation (FSV-ATG) tool is executed to perform the efficient generation of test cases. As a first step the input model is elaborated and automatically enriched with a set of test objectives to enable the subsequent test generation procedure for covering the state machine to obtain MC/DC coverage. The summary of the generation process for the cruise control example is shown in Fig. 7: a total number of 191 test objectives have been added to the formal model to enable the ATG step. The ATG activity is executed and as results generates a set of efficient test case. For the cruise control example the ATG generated 31 test cases given a bound of 100 maximum test cases and a depth bound of 30 steps. To validate the obtained results we simulated the generated test cases on the SUT evaluating the coverage value using the Simulink Verifcation and Validation (V&V) toolbox which provides an independent measure of effectiveness for our approach. The obtained independent measured coverage has been of 100% for decision, condition coverage and MC/DC coverage.

C. Additional experiments

In order to quantitatively compare the proposed algorithm and implementation with respect to state-of-the art tool we

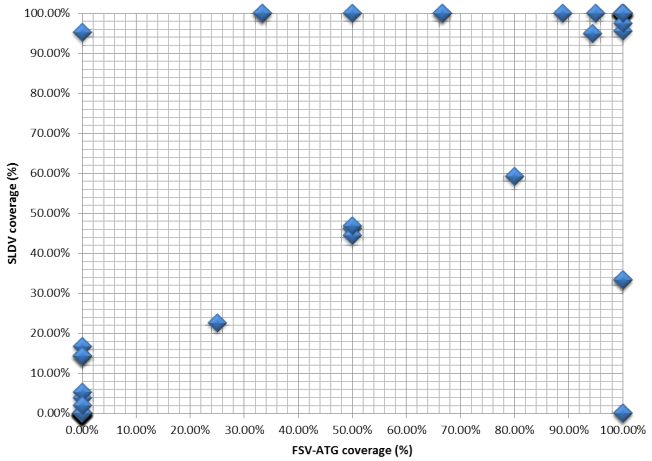

Fig. 8. Comparison of MCDC Coverage Results

compared the test suites generated using the Formal Specs Verifier ATG tool with the Simulink Design Verifier tool (SLDV) [2] that represents an industrial strength tool for the generation of test suites from MATLAB Simulink/Stateflow systems. A set of verification cases has been set up containing blocks ranging over a rich subset of Simulink/Stateflow Libraries. The results for the comparison of the MCDC coverage of the produced suites are presented in Fig. 8. The coverage values are obtained using the MATLAB VnV toolbox[3]. The analysis of the results shows that in some cases the SLDV tool is capable of achieving higher coverage with respect to FSV-ATG whereas in other cases the opposite is true. Cases where the MC/DC coverage percentage is 0% for FSV-ATG indicate that our algorithm was not able to generate any test cases. The analysis of the "losing" cases for the FSV-ATG tool showed that the low coverage is due to an inefficient translation of the Simulink/Stateflow block that does not allow efficient application of our ATG algorithm, as highlighted in Sec. VII-A. As an example, in some cases the FSV-ATG tool produces a structure of comparison/logical blocks that is inefficient for generating a high number of test objectives. The optimization of the translation step for achieving efficient coupling with the test generation engine is one of the activities we have planned as the next development steps of our tool.

VIII. Conclusion

Our model-based test generation algorithm produces a test suite starting from a model of the system under test (SUT) that is enriched with a set of test objectives to be satisfied by the test cases derived from a given coverage metric. The produced suite covers the model test objectives in an efficient way such that 1) there are no two test cases satisfying the same set of test objectives and 2) each test case covers the maximum number of test objectives for a given length. Our algorithm relies on the combination of bounded model checking with an optimization-based formulation of the test generation problem. The algorithm is implemented using an incremental execution approach that mitigates the complexity of the problem allowing successful application to complex industrial use cases. Several ways of algorithm-improvement are possible, though. From a technical standpoint the algorithm could be improved by employing parallelism at test generation level and not only at the formal back-end level. In addition, the last advancements in pseudo-Boolean SAT solving can be taken into account to explore additional formal back-ends. From a methodological standpoint the use of contract-based design ([14], [15]) could allow for exploiting a compositional approach at test generation level. From a methodological standpoint the use of contract-based design could allow for exploiting a compositional approach at test generation level. Finally, the use of formal proofs on the model can be used to ease the process of test generation.

[2]https://www.mathworks.com/simulinkdv

[3]http://www.mathworks.com/products/simverification/

References

[1] O. Ferrante, A. Ferrari, and M. Marazza, "Model based generation of high coverage test suites for embedded systems," in *European Test Symposium*, 2014.

[2] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 226–247, 2010.

[3] S. Rayadurgam and M. P. E. Heimdahl, "Generating mc/dc adequate test sequences through model checking." in *SEW*. IEEE Computer Society, 2003, p. 91.

[4] O. Ferrante, L. Benvenuti, L. Mangeruca, C. Sofronis, and A. Ferrari, "Parallel nusmv: A nusmv extension for the verification of complex embedded systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Ortmeier and P. Daniel, Eds. Springer Berlin Heidelberg, 2012, vol. 7613, pp. 409–416.

[5] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. London, UK, UK: Springer-Verlag, 1999, pp. 146–162.

[6] M. Kadono, T. Tsuchiya, and T. Kikuno, "Using the nusmv model checker for test generation from statecharts," in *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim International Symposium on*, 2009, pp. 37–42.

[7] G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, and M. Roveri, "Semi-formal functional verification by efsm traversing via nusmv," in *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*, 2010, pp. 58–65.

[8] G. Hamon, L. deMoura, and J. Rushby, "Generating efficient test sets with a model checker," in *2nd International Conference on Software Engineering and Formal Methods*. Beijing, China: IEEE Computer Society, Sep. 2004, pp. 261–270.

[9] http://www.mathworks.com/products/simulink/.

[10] J. Chilenski and S. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[11] M. Marazza, O. Ferrante, and A. Ferrari, "Automatic generation of failure scenarios for SoC," *ERTS*, 2014, February 5th.

[12] A. Ferrari, L. Mangeruca, O. Ferrante, and A. Mignogna, "DesyreML: a sysml profile for heterogeneous embedded systems," *ERTS, Embedded Real Time Software and Systems*, 2012.

[13] W. Aldrich, "Coverage analysis for model based design tools," *Proc. of the 18th International Conference and Exposition on Testing*, 2001.

[14] L. Mangeruca, O. Ferrante, and A. Ferrari, "Formalization and completeness of evolving requirements using contracts," in *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, 2013, pp. 120–129.

[15] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen, "Contracts for System Design," INRIA, Rapport de recherche RR-8147, Nov. 2012. [Online]. Available: http://hal.inria.fr/hal-00757488